

Contenido

PICKERS.....	101
TimePicker	101
DatePicker	102
DatePickerDialog	103
ALERT DIALOG	104
Construir tu propio diálogo	105
TAREAS ASÍNCRONAS	106
PROGRESSBAR	107
Problemas al rotar la pantalla	110
NOTIFICACIONES BARRA ESTADO	111
Elementos de una notificación	111
Estilo BigText.....	113
Estilo BigPicture.....	114
Estilo Inbox, línea a línea	115
Botones	115



6.

Interfaz de Usuario II

PICKERS

Android proporciona unos controles que ayudan al usuario a elegir la hora o la fecha que sea válida, con el formato correcto y ajustado a la configuración regional del usuario. Usando un picker nos aseguramos que la hora o fecha introducida sea válida, en un formato correcto y ajustado a la zona geográfica del usuario.

TimePicker

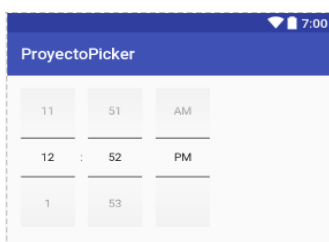
Permite la selección de la hora ya sea en formato AM/PM o en formato 24 horas. También tiene dos modos distintos de visualizarse, como *spinner* o modo *clock*.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.profesor.proyectopicker.MainActivity">

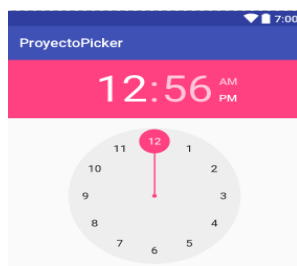
    <TimePicker android:id="@+id/simpleTimePicker"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:timePickerMode="spinner" />

</android.support.constraint.ConstraintLayout>
```

Con la propiedad `android:timePickerMode="spinner"` la apariencia del elemento es el siguiente:



Si el valor fuera *clock* la vista cambiaría:



Para visualizar en formato 24 horas lo haremos en código:

```
simpleTP = (TimePicker) findViewById(R.id.simpleTimePicker);
simpleTP.setIs24HourView(true);
```

Si queremos actuar cuando se producen cambios sobre la vista:

```
simpleTP.setOnTimeChangedListener(new TimePicker.OnTimeChangedListener() {
    @Override
    public void onTimeChanged(TimePicker view, int hourOfDay, int minute) {
        // display a toast with changed values of time picker
        Toast.makeText(getApplicationContext(), hourOfDay + " " + minute, Toast.LENGTH_SHORT).show();
    }
});
```

DatePicker

Un *DatePicker* o selector de fecha es un *view* elaborado para permitir al usuario seleccionar una fecha configurable y se puede elegir el año, el mes y el día respectivamente.

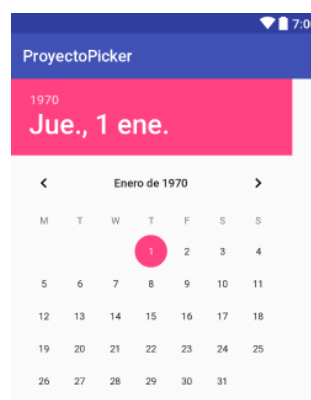
Tiene dos modos e visualización distintos *spinner* y *calendar*. La particularidad es que el primer modo visualiza tanto el *spinner* como el calendario.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.profesor.proyectopicker.MainActivity">

    <DatePicker android:id="@+id/simpleTimePicker"
        android:layout_width="368dp"
        android:layout_height="wrap_content"
        android:datePickerMode="calendar" />

</android.support.constraint.ConstraintLayout>
```

Su aspecto sería:



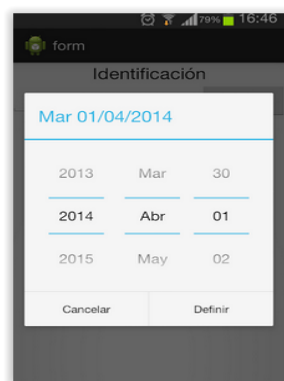
DatePickerDialog

Lo más frecuente es que estos controles no aparezcan directamente en la interfaz, sino que cuando pulsamos, por ejemplo, encima de un *EditText* nos aparezca la vista correspondiente a un *TimePicker* o *DatePicker* y unos botones que nos permitirán actuar sobre el control (crear un proyecto nuevo que se llamen proyectoDPdialog).

En el *layout* de la actividad principal solamente tendremos un *EditText* sobre el que implementaremos la funcionalidad siguiente: cuando pulsemos sobre él nos aparecerá un *DatePickerDialog* y cuando seleccionemos una fecha ésta aparecerá en el *EditText*.

```
public class MainActivity extends AppCompatActivity {
    EditText fecha;
    DatePickerDialog datePickerDialog;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        fecha=(EditText)findViewById(R.id.fechaNac);
        fecha.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // calendar class's instance and get current date , month and year from calendar
                final Calendar c = Calendar.getInstance();
                int mYear = c.get(Calendar.YEAR); // current year
                int mMonth = c.get(Calendar.MONTH); // current month
                int mDay = c.get(Calendar.DAY_OF_MONTH); // current day
                // date picker dialog
                datePickerDialog = new DatePickerDialog(MainActivity.this,
                    new DatePickerDialog.OnDateSetListener() {
                        @Override
                        public void onDateSet(DatePicker view, int year, int monthOfYear, int dayOfMonth) {
                            // set day of month , month and year value in the edit text
                            fecha.setText(dayOfMonth + "/"
                                + (monthOfYear + 1) + "/" + year);
                        }
                    }, mYear, mMonth, mDay);
                datePickerDialog.show();
            }
        });
    }
}
```



Podemos plantear algo similar con un *TimePickerDialog*.

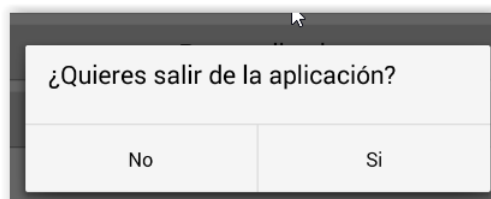
Otra forma de crear los *DatePickerDialog* es derivar de la clase *DialogFragment*, pero esto se verá más adelante.



ALERT DIALOG

Los *AlertDialog* son útiles para pedir confirmaciones, o bien formular preguntas que requieran pulsar “aceptar” o “cancelar”. Se basan en la clase *Dialog*.

```
AlertDialog.Builder builder = new AlertDialog.Builder(MyActivity.this);
builder.setMessage("¿Quieres salir de la aplicación?")
    .setCancelable(false)
    .setPositiveButton("Si", new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            MyActivity.this.finish();
        }
    })
    .setNegativeButton("No", new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            dialog.cancel();
        }
    });
AlertDialog alert = builder.create();
alert.show();
```



Si queremos una lista de la cual seleccionar la opción, podemos hacerlo de la siguiente manera:

```
final String[] items = {"Español", "Inglés", "Francés"};

AlertDialog.Builder builder =
    new AlertDialog.Builder(MyActivity.this);

builder.setTitle("Selección")
    .setItems(items, new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int item) {
            Log.i("Dialogos", "Opción elegida: " + items[item]);
        }
    });
builder.show();
```



También se pueden colocar checkboxes o radio buttons, consultar la documentación de Android.



Construir tu propio diálogo

Podemos crear nuestro propio diálogo personalizado, pasando la definición del mismo (el layout donde está definido) a `setContentView(View)`.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:padding="3dp" >

    <ImageView
        android:id="@+id/imageView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="3dp">

        <TextView
            android:id="@+id/textView1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/dialogo_linea_1" />

        <TextView
            android:id="@+id/textView2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/dialogo_linea_2" />

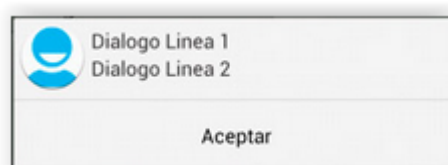
    </LinearLayout>
</LinearLayout>
```

En primer lugar se define el *context*, a continuación debemos vincular el diseño anterior con el cuadro de diálogo.

```
AlertDialog.Builder builder = new AlertDialog.Builder(MyActivity.this);

LayoutInflater inflater = MyActivity.this.getLayoutInflater();

builder.setView(inflater.inflate(R.layout.dialogo, null))
    .setPositiveButton("Aceptar", new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            dialog.cancel();
        }
    });
builder.show();
```



EjercicioPropuestoDiálogos



TAREAS ASÍNCRONAS

Un hilo es una unidad de ejecución asociada a una aplicación. Es la estructura de la programación concurrente, la cual tiene como objetivo dar la percepción al usuario que el sistema que ejecuta realiza múltiples tareas a la vez.

Cuando se construye una aplicación Android, todos los componentes y tareas son introducidos en el hilo principal o hilo de UI (*UI Thread*). Hasta el momento hemos trabajado de esta forma, ya que las operaciones que hemos realizado toman poco tiempo y no muestran problemas significativos de rendimiento visual en nuestros ejemplos.

Pero en tus proyectos reales no puedes pretender que todas las acciones que lleva a cabo tu aplicación sean simples y concisas.

En ocasiones hay instrucciones que toman unos segundos en terminarse (descarga de un archivo, carga de datos de una base de datos remota,...), esta es una de las mayores causas de la terminación abrupta de las aplicaciones.

La solución es ejecutar en segundo plano la otra actividad para continuar de forma fluida con la funcionalidad de la aplicación y evitar paradas inesperadas. Es aquí donde entran los hilos o la gestión de tareas asíncronas.

Como los hilos son menos eficientes que las tareas asíncronas vamos a centrarnos en estas últimas.

El objetivo de una tarea asíncrona es liberar al programador del uso de hilos, la sincronización entre ellos y la presentación de resultados en el hilo primario. Esta clase unifica los aspectos relacionados que se realizarán en segundo plano y además gestiona de forma asíncrona la ejecución de las tareas.

Para implementarla debes extender una nueva clase con las características de `AsyncTask` e implementar los métodos correspondientes para la ejecución en segundo plano y la publicación de resultados en el `UI Thread`.

AsyncTask una interfaz android.os que nos va a permitir crear un hilo secundario en el que realizar operaciones en background.

```
private class BucleProgressDialog extends AsyncTask<Void, Integer, Integer>
```

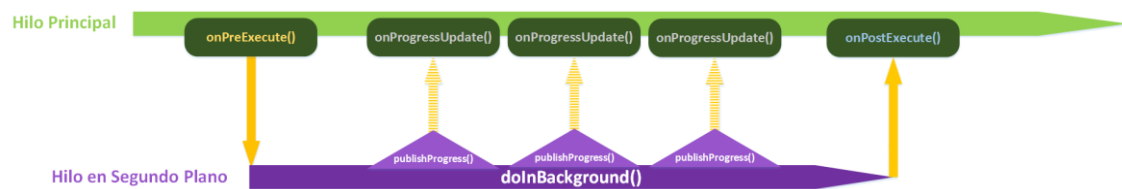
Esta interfaz tiene tres parámetros <T,T,T> que corresponden a:

El tipo de datos que recibiremos como **entrada** de la tarea en el método `doInBackground()`.

El tipo de datos con el que actualizaremos el **progreso** de la tarea, y que recibiremos como parámetro del método `onProgressUpdate()` y que a su vez tendremos que incluir como parámetro del método `publishProgress()`.

El tipo de datos que devolveremos como **resultado** de nuestra tarea, que será el tipo de retorno del método `doInBackground()` y el tipo del parámetro recibido en el método `onPostExecute()`.





Los métodos que se pueden implementar de esta interfaz son:

- **onPreExecute()**: en este método van todas aquellas instrucciones que se ejecutarán antes de iniciar la tarea en segundo plano. Normalmente es la inicialización de variables, objetos y la preparación de componentes de la interfaz.
- **doInBackground()**: es el único método obligatorio de implementar dentro de la interfaz. Será el encargado de realizar la tarea en segundo plano y recibe los parámetros de entrada para la misma. Se ejecuta tras el *onPreExecute()*. Dentro de él podemos invocar un método auxiliar llamado *publishProgress()*, el cual transmitirá unidades de progreso al hilo principal.
- **onProgressUpdate()**: recibe los datos del método anterior a través de *publishProgress()*. Se ejecuta en la interfaz del hilo principal permitiendo a través de una vista la actualización del *progressDialog* de forma que el usuario puede saber cómo avanza el progreso.
- **onPostExecute()**: se ejecuta en último lugar, en él cerraremos el *progressDialog*.
- **onCancelled()**: se ejecutará cuando se cancele la ejecución de la tarea antes de su finalización normal.

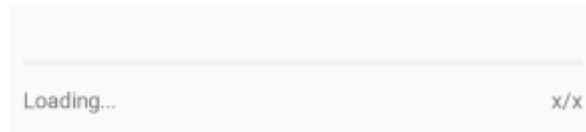
Estos métodos tienen una particularidad esencial para nuestros intereses. El método *doInBackground()* se ejecuta en un hilo secundario (por tanto no podremos interactuar con la interfaz), pero sin embargo todos los demás se ejecutan en el hilo principal, lo que quiere decir que dentro de ellos podremos hacer referencia directa a nuestros controles de usuario para actualizar la interfaz. Por su parte, dentro de *doInBackground()* tendremos la posibilidad de llamar periódicamente al método *publishProgress()* para que automáticamente desde el método *onProgressUpdate()* se actualice la interfaz si es necesario.

PROGRESSBAR

A partir de la API 26 el *progressDialog* está obsoleto. En su sustitución entra en juego la *progressBar*. La barra de progreso se define dentro de un archivo XML y posteriormente se implementa a través de un *dialog* (creamos un nuevo proyecto que se llame *ProgressBar*).



En primer lugar definimos el aspecto que tendrá nuestra barra de progreso, para ello definimos el siguiente archivo XML con el nombre *dialogo_progreso*:



```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:padding="13dp"
    android:id="@+id/Dialogoprogreso"
    android:layout_centerHorizontal="true"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <ProgressBar
        android:id="@+id/loader"
        style="@style/Widget.AppCompat.ProgressBar.Horizontal"
        android:layout_width="match_parent"
        android:layout_height="65dp" />

    <TextView
        android:layout_width="wrap_content"
        android:text="Loading..."
        android:layout_alignBottom="@id/loader"
        android:textAppearance="?android:textAppearanceSmall"
        android:id="@+id/loading_msg"
        android:layout_alignParentLeft="true"
        android:layout_height="wrap_content" />

    <TextView
        android:layout_width="wrap_content"
        android:text="x/x"
        android:layout_alignBottom="@id/loader"
        android:textAppearance="?android:textAppearanceSmall"
        android:layout_gravity="center_vertical"
        android:id="@+id/progres_msg"
        android:layout_alignParentRight="true"
        android:layout_height="wrap_content" />
</RelativeLayout>
```

En la ventana principal de nuestra aplicación tendremos un botón que cuando se pulse iniciará la ejecución de la tarea y se visualizará la barra de progreso mientras dure la misma.

Para simular esta tarea implementaremos una tarea asíncrona con un bucle y unos retardos.



```

public class MainActivity extends AppCompatActivity {
    Dialog dialog;
    ProgressBar progressBar;
    int progreso;
    TextView loadingMessage, progressMessage;
    TareaAsincrona tareaAsincrona;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        findViewById(R.id.boton).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                tareaAsincrona=new TareaAsincrona();
                tareaAsincrona.execute(0);
            }
        });
    }
}

```

Veamos el código de la tarea asíncrona:

```

class TareaAsincrona extends AsyncTask<Integer,Integer,Boolean> {
    @Override
    protected void onPreExecute() {
        setDialog();
    }

    @Override
    protected Boolean doInBackground(Integer... values) {
        try {
            for(int i=values[0]; i<=100 ; i++) {
                Thread.sleep(100);
                publishProgress(i);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return true;
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        super.onProgressUpdate(values);
        progreso = values[0].intValue();

        progressBar.setProgress(progreso);
        progressMessage.setText(values[0]+"/100");
    }

    @Override
    protected void onPostExecute(Boolean result) {
        super.onPostExecute(result);
        if(result) {
            Toast.makeText(MainActivity.this, "Tarea finalizada!",
                Toast.LENGTH_SHORT).show();
            dialog.dismiss();
        }
    }
}

```



En el método *onPreExecute* definimos o inicializamos el diálogo:

```
private void setDialog(){
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    View view=getLayoutInflater().inflate(R.layout.dialogo_progress,null);
    progressBar=view.findViewById(R.id.loader);
    loadingMessage=view.findViewById(R.id.loading_msg);
    progressMessage=view.findViewById(R.id.progres_msg);
    progressBar.setMax(100);
    builder.setView(view);
    dialog = builder.create();
    dialog.setCancelable(false);
    dialog.show();
}
```

En el método *doInBackground* implementamos la tarea que en nuestro caso la simulamos con un bucle de 100 iteraciones con un retardo en cada una de ellas (suponer que es la descarga de una imagen desde un servidor remoto). En este método ejecutamos *publishProgress* para que se pueda realizar la actualización de la barra de progreso en el método *onProgressUpdate*.

Cuando finaliza la tarea visualizamos un *toast* con esa indicación.

Problemas al rotar la pantalla

Si estamos ejecutando una tarea asíncrona y giramos la pantalla la activity se reinicia y la tarea asíncrona no se reanuda. Esto en sí mismo es un problema, porque lo interesante sería que la tarea asíncrona (con la barra de progreso) continuara ejecutándose desde el momento justo en el que el dispositivo gira.

Para poder controlar esta circunstancia debemos implementar los métodos que van a permitirnos almacenar o recuperar los estados en el momento del giro *onSaveInstanceState* y *onRestoreInstanceState*.

```
//Métodos para salvar el estado de la barra de progreso (o cualquier otra cosa) cuando pasa por
//onStop (es decir girando la pantalla)
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    tareaAsincrona=new TareaAsincrona();
    tareaAsincrona.execute(savedInstanceState.getInt("VALUE"));
}
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    if(progressBar!=null && tareaAsincrona!=null ){
        outState.putInt("VALUE",progressBar.getProgress());
        tareaAsincrona.cancel(true);
        dialog.dismiss();
    }
}
```

Evidentemente si la aplicación entra en pausa por decisión propia del usuario la activity y la tarea asíncrona son interrumpidas, para poder controlar esta circunstancia debemos implementar los métodos *onResume()* y *onPause()* en la activity principal.



```

//Métodos para salvar el estado de la barra de progreso (o cualquier otra cosa) cuando pasa por
//onPause (es decir pulsando boton derecha cuadrado)
@Override
protected void onResume() {
    super.onResume();
    if(tareaAsincrona!=null && tareaAsincrona.isCancelled())
    {
        tareaAsincrona=new TareaAsincrona();
        tareaAsincrona.execute(progreso);
    }
}
@Override
protected void onPause() {
    super.onPause();
    if(tareaAsincrona!=null && progressBar!=null){
        progreso=progressBar.getProgress();
        tareaAsincrona.cancel(true);
        dialog.dismiss();
    }
}
}

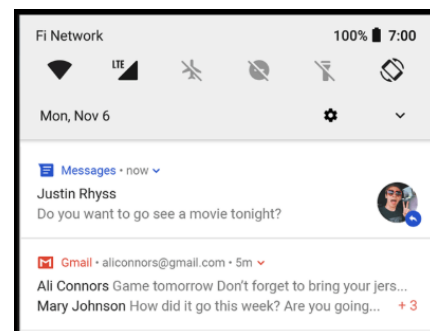
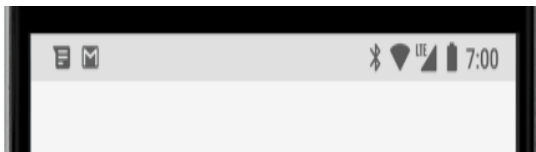
```

Ejercicio Propuesto Cronómetro

NOTIFICACIONES BARRA ESTADO

La barra de estado de Android se encuentra situada en la parte superior de la pantalla. La parte izquierda de esta barra está reservada para visualizar notificaciones. Cuando se crea una nueva notificación, aparece un texto desplazándose en la barra, y a continuación, un pequeño icono permanecerá en la barra para recordar al usuario la notificación.

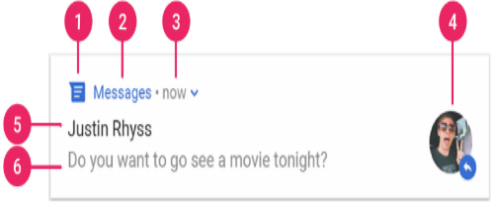
El usuario puede arrastrar la barra de notificaciones hacia abajo, para mostrar el listado de las notificaciones por leer.



Elementos de una notificación

Las notificaciones podemos verlas de dos formas distintas, la vista normal antes de que el usuario la despliegue y la vista ampliada.



<p style="text-align: center;">Notificación con vista normal</p> 	<ol style="list-style-type: none"> 1. Icono pequeño: se establece con setSmallIcon(). 2. Nombre de la aplicación: proporcionado por el sistema. 3. Hora a la que se emitió la notificación. Se puede establecer un valor explícito con setWhen() u ocultarlo setShowWhen(false); si no se hace se mostrará por defecto la hora del sistema en el momento de recepción de la notificación. 4. Icono grande: es opcional (generalmente se usa solo para fotos de contacto; no lo use para el icono de su aplicación) se establece con setLargeIcon(). 5. Título: Esto es opcional y se establece con setContentTitle(). 6. Texto: esto es opcional y se establece con setContentText().
---	---

La vista ampliada sólo aparece cuando se expande la notificación, lo que sucede cuando la notificación está en la parte superior del buzón de notificaciones, o cuando el usuario amplía la notificación con un gesto.

Canales de notificación

A partir de Android 8.0 (nivel de API 26), todas las notificaciones deben asignarse a un canal o directamente no serán visibles. Los canales de notificación permiten a los desarrolladores agrupar las notificaciones en categorías (canales). Esto permitirá al usuario la habilidad de modificar los ajustes de notificación para el canal entero a la vez. Por ejemplo, para cada canal, los usuarios pueden bloquear completamente todas las notificaciones, anular el nivel de importancia, o permitir que la insignia de la notificación se muestre. Los usuarios también pueden presionar una notificación para cambiar los comportamientos del canal asociado. Todas las notificaciones publicadas en el mismo canal de notificación tienen el mismo comportamiento.

Nota: la interfaz de usuario se refiere a los canales de notificación como "categorías".

En los dispositivos que ejecutan Android 7.1 (nivel de API 25) e inferior, los usuarios pueden administrar las notificaciones solo por aplicación (de hecho, cada aplicación solo tiene un canal).

Por lo tanto antes crear la notificación, deberemos crear el canal o los canales a los que queremos añadir nuestras notificaciones.

Crear canales de notificación

Antes de publicar una notificación se debe crear el canal, por lo que es importante que al iniciarse la aplicación o antes de crear la notificación se ejecute el código de creación. Veamos un ejemplo para crear dos canales de notificación:



```

private NotificationChannel crearCanal(String idCanal,String nombreCanal, String descripcion, int importancia )
{
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        NotificationChannel canal = new NotificationChannel(idCanal, nombreCanal, importancia);
        canal.setDescription(descripcion);
        return canal;
    }
    return null;
}

private void crearCanalesNotificacion() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        NotificationManager notificationManager = getSystemService(NotificationManager.class);
        NotificationChannel canal = crearCanal(CHANNELAVISOS_ID, "Avisos", "Avisos importantes", NotificationManager.IMPORTANCE_HIGH);
        canal.setVibrationPattern( new long[] {400,600,100,300,100});
        // Registrando el canal en el sistema. Después de esto no se podrá cambiar
        // las características del canal (importancia u otras propiedades del)
        notificationManager.createNotificationChannel(canal);
        canal = crearCanal(CHANNELMENSAJES_ID, "Mensajes", "Mensajes", NotificationManager.IMPORTANCE_LOW);
        notificationManager.createNotificationChannel(canal);
    }
}

```

Los pasos a seguir son:

1. Construir un objeto del tipo **NotificationChannel**, con un ID de canal único, un nombre visible para el usuario, una descripción y un nivel de importancia.
2. Si en el constructor no especificamos la descripción, se puede especificar después con **setDescription()**.
3. Podemos aplicar otros comportamientos visuales y sonoros a nuestro canal, en el ejemplo hemos activado vibración
4. Por último registrar el canal de notificación pasándolo al sistema con el método **createNotificationChannel()**.

En nuestro ejemplo definimos un método que crea un canal y que es llamado por el método que asigna los canales creados a la aplicación.

La prioridad de nuestras notificaciones se ajusta a las siguientes opciones:

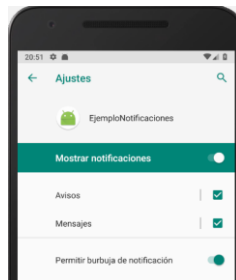
User-visible importance level	Importance (Android 8.0 and higher)	Priority (Android 7.1 and lower)
Urgent Makes a sound and appears as a heads-up notification	IMPORTANCE_HIGH	PRIORITY_HIGH or PRIORITY_MAX
High Makes a sound	IMPORTANCE_DEFAULT	PRIORITY_DEFAULT
Medium No sound	IMPORTANCE_LOW	PRIORITY_LOW
Low No sound and does not appear in the status bar	IMPORTANCE_MIN	PRIORITY_MIN

Evidentemente, a todas las notificaciones para un canal se les dará el mismo nivel de importancia.

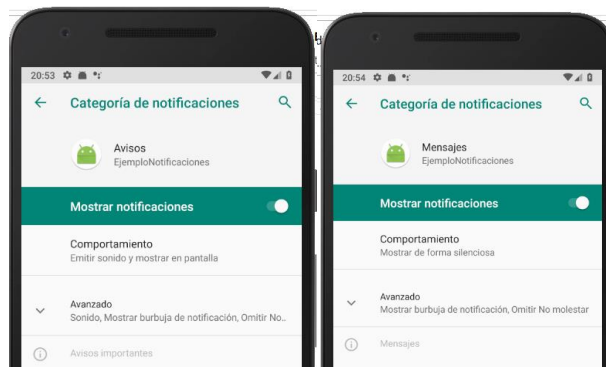
Las categorías creadas aparecerán en las opciones de configuración de nuestro dispositivo.



Ir a Ajustes->Notificaciones->Seleccionamos Aplicación y nos aparecerá la categoría creada. También se mostrará si se pulsa largamente sobre la notificación desplegada y después sobre el icono de información.



Si entramos dentro de la categoría veríamos su definición detallada:



Es en este panel donde el usuario activa o no las notificaciones de una determinada categoría, también puede cambiar la acción que ocurrirá al lanzarse la notificación asociada a la categoría.

Crear una notificación

Para crear notificaciones se utiliza la clase `NotificationCompat.Builder`. Para crear una notificación solamente crearemos un objeto de este tipo y le asignaremos el contexto de la aplicación.

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(mContext, ID_Channel);
```

Y se usa como en el siguiente ejemplo:

```
NotificationCompat.Builder notificacion;  
notificacion=new NotificationCompat.Builder(getApplicationContext(),CHANNELMENSAJES_ID);  
notificacion.setContentText("Este es el texto de mi notificación");  
notificacion.setContentTitle("Mi notificación");  
notificacion.setSmallIcon(R.mipmap.ic_launcher);  
notificacion.setLargeIcon(((BitmapDrawable) ContextCompat.getDrawable(this,R.mipmap.ic_launcher)).getBitmap());  
notificacion.setTicker("Optional ticker");  
notificacion.setWhen(System.currentTimeMillis());  
notificacion.setAutoCancel(true);  
//elimina notificación una vez visualizada
```



Los métodos `setContentTitle()` y `setContentText()`, permiten colocar el título y el texto de la notificación. Para mostrar los iconos se utilizarán los métodos `setSmallIcon()` y `setLargeIcon()` que se corresponden con los iconos mostrados a la derecha y a la izquierda del contenido de la notificación en versiones recientes de Android, en versiones más antiguas tan sólo se mostrará el icono pequeño a la izquierda de la notificación. Además, el icono pequeño también se mostrará en la barra de estado superior.

El ticker (texto que aparece por unos segundos en la barra de estado al generarse una nueva notificación) lo añadiremos mediante `setTicker()` y el texto auxiliar (opcional) que aparecerá a la izquierda del icono pequeño de la notificación mediante `setContentInfo()`.

La fecha/hora asociada a nuestra notificación se tomará automáticamente de la fecha/hora actual si no se establece nada, o bien puede utilizarse el método `setWhen()` para indicar otra marca de tiempo.

El siguiente paso, una vez tenemos completamente configuradas las opciones de nuestra notificación, será el de generarla llamando al método `notify()` del *Notification Manager*, al cual podemos acceder mediante una llamada a `getSystemService()` con la constante `Context.NOTIFICATION_SERVICE`. Por su parte al método `notify()` le pasaremos como parámetro un identificador único definido por nosotros que identifique nuestra notificación y el resultado del builder que hemos construido antes, que obtenemos llamando a su método **`getNotification()`**, o con `build()` si estamos en API 16 o superior.

```
NotificationManager mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);  
mNotificationManager.notify(0, notificacion.build());
```

El último paso será establecer la actividad a la cual debemos dirigir al usuario automáticamente si éste pulsa sobre la notificación. Para ello debemos construir un objeto `PendingIntent`, que será el que contenga la información de la actividad asociada a la notificación y que será lanzado al pulsar sobre ella. Para ello definiremos en primer lugar un objeto `Intent`, indicando la clase de la actividad concreta a lanzar, que en nuestro caso será la propia actividad principal de ejemplo (`MainActivity.class`). Este *intent* lo utilizaremos para construir el `PendingIntent` final mediante el método `PendingIntent.getActivity()`. Por último asociaremos este objeto a la notificación mediante el método `setContentIntent()` de la notificación creada.

Veamos cómo quedaría esta última parte comentada:

```
Intent intent2=new Intent(MainActivity.this,MainActivity.class);  
PendingIntent pendingIntent=PendingIntent.getActivity(MainActivity.this, 0, intent2, 0);  
notificacion.setContentIntent(pendingIntent);
```

Pero además podemos añadir más funcionalidad si lanzamos otro `pendigintent` una vez borrada la notificación. Este `pending` lo añadiremos con el método **`setDeleteIntent()`** sobre la notificación, y se ejecutará al deslizar la notificación para eliminarla.



Las notificaciones nos permiten personalizarlas creando layouts propios, pero nosotros nos centraremos en los 3 estilos que vienen de serie. Recordad que estos estilos se expanden y contraen, por lo que tenemos que definir unos datos para la notificación normal y otros para la grande. Además, por defecto sólo aparecerá expandida la primera notificación de la bandeja, el resto estarán contraídas y el usuario podrá abrirlas manualmente con un gesto. Los estilos posibles son:

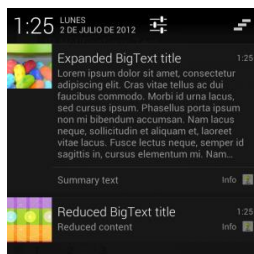
Estilo *BigText*

Se usa a partir de un objeto de tipo Builder, pasándolo como parámetro a la clase **NotificationCompat.BigTextStyle** para poder aplicarle algunos métodos opcionales. Queda más claro con un ejemplo:

```
NotificationCompat.BigTextStyle n= new NotificationCompat.BigTextStyle(notificacion)
    .bigText("Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec id ex hendrerit, ullamcorper")
    .setBigContentTitle("Expanded BigText title")
    .setSummaryText("Summary text");
mNotificationManager.notify(2, n.build());
```

Como se puede suponer al crear el estilo, lo haremos sobre una notificación base creada con anterioridad y que la pasaremos al objeto BigTextStyle.

Al crear el nuevo estilo estableceremos el **texto largo**, como puede ser el contenido de un correo electrónico o de un SMS, el título que llevará en su forma expandida si queremos que sea diferente, y un resumen opcional que se mostrará en 1 línea al final del texto. En la captura podéis ver un ejemplo de cómo quedaría la misma notificación expandida y contraída (la imagen no coincide exacta con el código).



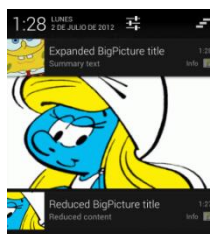
Estilo *BigPicture*

Si la notificación es relativa a una imagen también podemos usar una notificación expandida para mostrar dicha imagen en la bandeja gracias a **NotificationCompat.BigPictureStyle**. Su uso es casi idéntico al anterior, cambiando la clase y métodos del estilo.

```
NotificationCompat.BigPictureStyle n2= new NotificationCompat.BigPictureStyle(notificacion)
    .bigPicture(((BitmapDrawable) ContextCompat.getDrawable(this, R.mipmap.ic_launcher)).getBitmap())
    .bigLargeIcon(((BitmapDrawable) ContextCompat.getDrawable(this, R.mipmap.ic_launcher)).getBitmap())
    .setBigContentTitle("Expanded BigPicture title")
    .setSummaryText("Summary text");
```



En este caso en vez de colocarle un texto le ponemos la **imagen grande** que queremos mostrar con **bigPicture()** y si queremos que la imagen “pequeña” a la izquierda de la notificación sea distinta en la forma expandida también la cambiamos con **bigLargeIcon()**. En el ejemplo he decidido que la imagen de la notificación en su forma contraída será la misma que la imagen grande en la forma expandida, en cuyo caso la imagen a la izquierda será otra, por ejemplo el icono de la aplicación. Para muestra un botón.

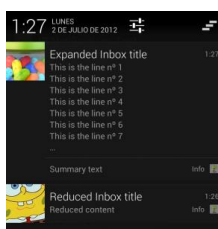


Estilo Inbox, línea a línea

El último estilo que tenemos es el **Notification.InboxStyle**, que muestra una lista de texto. Como varios correos recibidos, o lo que nos escribe alguien por mensajería instantánea. Muy similar también a los anteriores.

```
String[] lines={"Mensaje1","Mensaje2","Mensaje3","Mensaje4","Mensaje5"};
Notification.InboxStyle n3 = new Notification.InboxStyle(notificacion)
    .setBigContentTitle("Expanded Inbox title")
    .setSummaryText("Summary text");
for(String line : lines){
    n3.addLine(line);
}
mNotificationManager.notify(3, n3.build());
```

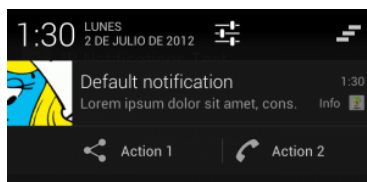
En esta ocasión añadimos las líneas una a una con **addLine()**, aunque nos aconsejan un máximo de 5. Nos quedaría algo así (código e imagen no coinciden).



Botones

Otra novedad introducida en Jelly Bean es la posibilidad de añadir **hasta 3 botones** a las notificaciones con sus acciones asociadas. Se hace sobre el objeto de tipo **Notification.Builder** con el método **addAction()**, así que es compatible con cualquiera de los 4 tipos de notificaciones anteriores (el tradicional no). En la notificación normal los botones se considerarán como parte “expandida”, por lo que tendrán el mismo comportamiento que las notificaciones grandes.





En el ejemplo vemos un método que añade 2 botones a cualquier builder que pasemos como parámetro.

```
NotificationCompat.Action action= new NotificationCompat.Action.Builder(android.R.drawable.ic_menu_delete,
                                "Delete", PendingIntent.getActivity(MainActivity.this, 0, i, 0)).build();
notificacion.addAction(action);
notificacion.addAction(android.R.drawable.ic_menu_share, "Share", PendingIntent.getActivity(MainActivity.this, 0, i, 0));
```

Le pasamos como parámetros un drawable del icono que llevará el botón a la izquierda del texto, el texto que llevará, y un PendingIntent que se activará cuando pulsemos el botón.

Algunos extras

Por ahora todo lo que hemos visto ha sido para crear la notificación, pero aún tenemos que mostrarla y quizá hacer que el móvil suene y/o vibre. También podemos hacer estas cosas con los métodos de la clase NotificationCompat.Builder. Veamos unos ejemplos

Sonido

Podemos reproducir un sonido al activar la notificación. El la imagen se reproduce el sonido por defecto, puede ser otro.

```
Uri defaultSound = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
notificacion.setSound(defaultSound);
```

Vibración

También podemos hacer que el teléfono vibre. Para eso lo primero que necesitamos es declarar el permiso de vibración `<uses-permission android:name="android.permission.VIBRATE"/>`

Luego creamos un patrón de vibración. Consiste en un array de long alternando el tiempo de vibración y pausa en milisegundos. Y se lo asignamos a la notificación.

```
// Patrón de vibración: 1 segundo vibra, 0.5 segundos para, 1 segundo vibra
long[] pattern = new long[]{1000,500,1000};
notificacion.setVibrate(pattern);
```

La vibración por defecto la podemos conseguir de la siguiente manera:

```
//Vibración por defecto
notificacion.setDefaults(Notification.DEFAULT_VIBRATE);
```



getActiveNotifications

A partir del Api 23 se ha agregado a la clase NotificationManager el método **getActiveNotifications()**. Muy útil para consultar el estado de las notificaciones, este método devuelve un array con el estado de todas las notificaciones activas en el momento.

StatusBarNotification [] getActiveNotifications ()

Recupera una lista de notificaciones activas: aquellas publicadas por la aplicación que aún no han sido descartadas por el usuario o canceladas por la aplicación. Se puede identificar la notificación a partir del tag o del id suministrado al notify(), así como una copia del original Notificationobjeto a través getNotification().

-  **Ejercicio Propuesto Notificaciones**
-  **Ejercicio Resuelto TemporizadorHuevo**

